

A General Architecture for Heterogeneous Language Engineering and Projectional Editor Support

Tony Clark

School of Science and Technology, Middlesex University, London, UK
 t.n.clark@mdx.ac.uk

Abstract. Tool support for language engineering has typically prioritises concrete syntax over abstract syntax by providing meta-languages for expressing concrete syntax and then mapping concrete to abstract structures. Text-based languages are usually specified using a BNF-like language used to generate a syntax-aware editor that includes features such as keyword completion. Similarly, graphical languages are defined using a declarative graphical syntax language, producing an editor that supports features such as shapes, graphs and edges. Projectional editors invert traditional approaches by prioritising abstract over concrete syntax. This paper describes a projectional meta-tool architecture, including general purpose abstract and concrete meta-languages, that uses declarative rules to integrate the syntax and tool support for a range of heterogeneous languages. The architecture has been implemented in Racket and the paper illustrates the architecture with concrete examples.

1 Introduction

Domain Specific Languages (DSLs) [28, 20] are motivated by the need to define languages that match specific use-cases, as opposed to General Purpose Languages (GPLs). Whilst GPLs are usually supported by standard text editors, DSLs, by their nature, often contain a range of more exotic syntax elements that are arguably better supported by syntax-aware editors. This has led to the development of a range of technologies to support DSL development and that generate tools for each DSL. Where the DSL is limited to text, languages such as EMFText [10], MontiCore [18, 19], TCS [13], and XText [8] and Spoofax [14] allow a DSL to be quickly and conveniently defined and the associated tooling generated. These technologies are mainly based on grammar-ware [15] that integrate language parsers with editors in order to achieve a workbench. Many of the technologies integrate static and dynamic analysis of the resulting DSL. These technologies have become quite mature and the term Language Workbench [9] has been coined to describe this type of engineering tool.

Whilst languages used for programming or scripting tend to be exclusively text-based, modelling languages have included a much wider palette of elements. UML for example, has a number of sub-languages that are based on graphs, but also includes text in the form of OCL and action languages. Relatively few technologies support the definition and tooling of DSLs containing graphical syntax elements. Notable exceptions are Eugenia [16, 17], GMF [11], MetaEdit+ [26]. However, these technologies do not support multi-mode languages where text and graphics can be freely mixed. In addition they do not have intrinsic support for defining semantics, particularly operational semantics, for the languages that are defined, and require the language engineer to step out of the system in order to provide such information.

There has been increasing attention to mixing graphical and textual notations [1, 24, 7, 23]. A recent model-based approach to mixing text and graphical languages is described in [2] that uses projectional editing techniques over a model. Whilst most of the reported work agrees on the general principles and propose approaches, this paper appears to be the first to

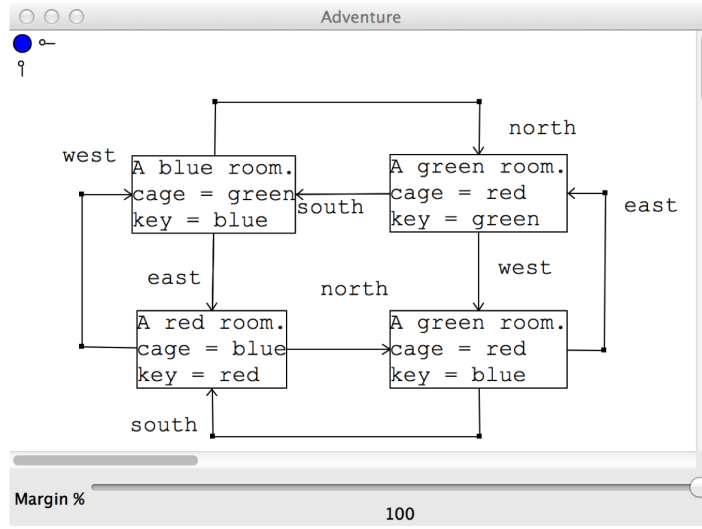


Fig. 1: Defining the Game

propose concrete meta-language that integrates with a general architecture for projectional editing in order to integrate textual and graphical languages.

Language engineering technologies such as those above tend to focus on concrete-syntax as the primary consideration. The language is often defined as a grammar or a collection of graphical elements that are then mapped to abstract-syntax structures. This is usually a one-to-one relationship. Projectional editors invert the traditional syntax dominance to promote abstract-syntax over concrete-syntax. The benefit of this is that the language definition does not dictate a single user experience since a single abstract-syntax can be mapped to multiple concrete-syntaxes depending upon mode, or execution state, or viewpoint.

The disadvantage of the projectional approach is that the supporting technology is less mature and less widespread. A small number of implementation platforms exist, such as MPS [29,30] and Intentional Software [25]. Whilst both of these technologies have been several years in development, there is no basic definition of a *projectional editor* or a framework for exploring variations.

This paper presents a simple architecture for a projectional editor and identifies several key features that should be supported including syntax representation, mappings and identity management. It goes further by proposing a simple meta-language that can be used to populate and drive such an architecture. The meta-language uses pattern-directed rules to transform syntax trees. It is based on the transformation rules of Stratego/XT [3], and is a distant cousin to model and language transformation engines such as ATL [12] and TXL [6] but extends these in terms of expressive power, and proposes two different categories of mapping: *transformations* and *reductions*. Both the architecture and meta-language have been implemented in Racket and the approach is demonstrated through a number of concrete running examples.

2 Example

Consider a game that involves a collection of rooms that are connected by corridors. A room is either empty or contains a locked cage. The cage is painted red, green or blue. Inside the cage is a painted key. A key can be used to unlock a cage of the same colour and get the

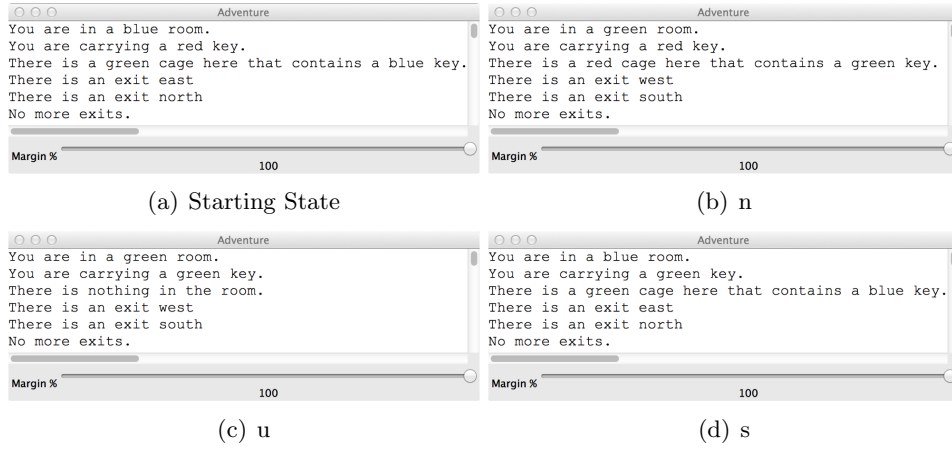


Fig. 2: Playing the Game

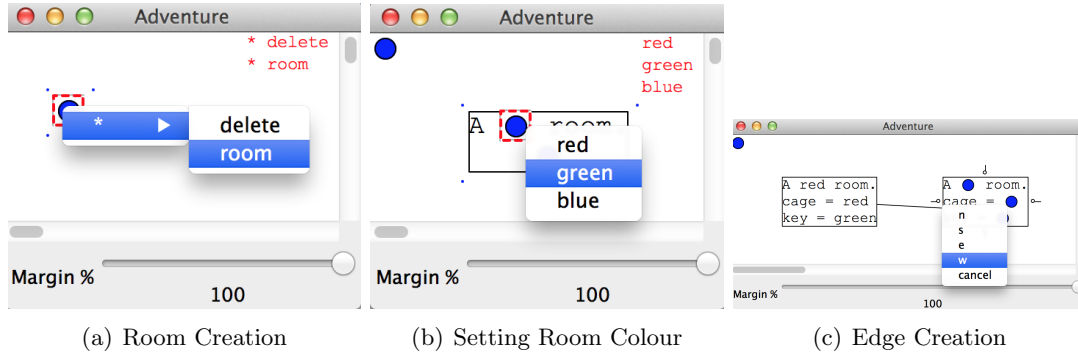


Fig. 3: Editor Interactions

key inside. The player starts off in a room with a red key. The aim of the game is to visit all the rooms and unlock all the cages. Figure 1 shows the definition of a dungeon using the language editor for game construction. Rooms are created as nodes and corridors as labelled edges. The text in a room-node shows the colour of the room, the colour of a cage and the colour of the key in the cage. The blue dot at the top-left corner of the tool is used to access a room-creation menu. Edges between room-nodes are created by dragging the mouse from a source node to the target (a menu is used to select a direction label). When a room-node is created, its colour and contents are unset. The mouse is used to select from pre-defined colours for the room, cage and key.

The language operates in two modes: creation and play, it is possible to switch between the modes by pressing `p` and `c` on the keyboard. Figure 2 shows play mode. The player starts in the blue room with a red key. The player makes a move by pressing the first letter of the direction on the keyboard. Since the player does not have a green key they must move from the starting room; they press `n` to go north and arrive at a green room with a red cage. The player can open the cage since their key matches the cage colour. This is done by pressing `u` on the keyboard. Finally, the player goes back south.

The game shows a number of features of the projectional editor. Interaction with the language can be moded; in this example there are two modes, but in general there can be any number. Figure 1 shows that the abstract syntax can be projected on to graphs and text. In addition, language features can be created by menus made available as blue-dots. Figure 1 shows a blue dot that is used to create room-nodes, but in general a language may offer many

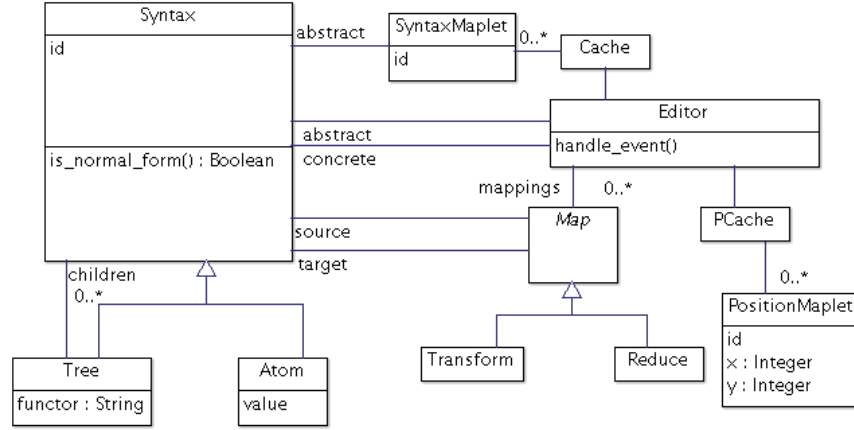


Fig. 4: The Editor Architecture

different types of item. Figure 2 shows that the state of the game is projected to become formatted text. Figure 3 shows how the editor that is generated from the language definition supports creation of language elements: (a) creation of a new room element; (b) selection of a room colour; (c) selection of a type of edge between rooms.

3 An Architecture for Projectional Editors

A projectional editor provides facilities for defining and constructing abstract syntax, reducing the abstract syntax to concrete syntax and then mapping user actions that are applied to the concrete syntax into transformations applied to the abstract syntax. The characteristic feature of the closure of the loop:

```

loop {
  concrete := reduce(abstract);
  display(concrete);
  wait(concrete-event) {
    abstract := transform(concrete-event, abstract);
  }
}

```

Figure 4 shows the proposed architecture for an editor that supports such a loop. Both concrete and abstract syntax can be implemented as a simple tree structure where the type of the structure is encoded as a functor at the root of each sub-tree. Abstract syntax will depend on the particular language being defined, however the concrete syntax must conform to a particular interface that is supported by the editor in order that it can display the syntax and can listen for user events. Therefore, **Syntax** defines a predicate `is_normal_form()` that is true when the syntax is in a format suitable for the user.

The loop shown above must map concrete events to actions that can be applied to the abstract syntax. In addition, each time round the loop, the abstract syntax produces a *different* concrete syntax structure. In both cases it must be possible to keep track of the association between abstract and concrete syntax elements. In the first case the association is used to apply the action to the correct abstract syntax element, and in the second case the user may have changed the concrete syntax, for example by moving graph nodes, and the new concrete syntax element must be associated with a current position.

The architecture shown in figure 4 uses identities on syntax elements in order to implement a mapping between abstract and concrete syntax. A fresh identity is coined when a

```

lang = (deflang id
  (abstract (id g-element ...))
  (locals local-def ...)
  (transform rule ...)
  (reduce rule ...))
g-element = id | str
  | (id g-element ...)
  | (* g-element ...)
  | (or g-element ...)
local-def = (id exp)
  | ((id pattern ...) exp)
rule = (pattern exp)
pattern = id | string | int | char | bool
  | (id pattern ...)
  | ((id pattern ...) pattern ...)
  | pattern dots
exp = id | string | int | char | bool
  | (id exp ...)
  | ((id exp ...) exp ...)
  | exp dots
  | (case exp rule ...)

nf = (nl) | string | int | bool | char
  | (font +/- nf)
  | (seq nf ...)
  | (tab nf)
  | (indent int nf)
  | (box bb bm (pos nf) ...)
  | (vbox bb bm (pos nf) ...)
  | (hbox bb bm (pos nf) ...)
  | (ellipse int int int int bool bool)
  | (rectangle int int int int bool bool)
  | (image int int string)
  | (underline nf)
  | (chars string)
  | (thumbnail int int int int)
  | (tree nf ...+)
  | (graph
    (edge-types (id (id) (id)) ...)
    g-node ...
    g-edge ...)
+/- = (+) | (-)
bb = (border int)
bm = (menu (id (id nf ...)) ...)
pos = left | right | top | bot | centre | align
g-node = (node (id) nf nf)
g-edge = (edge nf nf e-dec nf e-dec (label end nf))
e-dec = (arrow) | (none)
end = (source) | (target)

```

Fig. 5: The Language-Definition Language

new abstract element is created and may be passed on to individual concrete elements via the mapping. The editor maintains two caches. The first associates identities with abstract elements in order to support the event to action mapping. The second associates concrete identities with positional information.

The editor provides two types of mapping. The first is a transformational mapping that is used to change the abstract syntax tree. An example of such a transformation is the creation of a new node or editing a string. There is no requirement that the result of such a mapping is in normal form. The second type of mapping is a reduction that takes abstract syntax and produces syntax `t` such that `t.is_normal_form()`.

The architecture described above has been implemented as a tool in Racket. The editor is driven by a meta-language that defines the structure of the syntax trees and implements the mappings using pattern matching over trees. The rest of this paper describes the meta-language and provides a number of examples of its use that exemplify key features. The paper concludes with the implementation of the game described in section 2 and a discussion of possible extensions to the architecture.

4 A Meta-Language Definition Language

The language architecture described in section 3 is defined with respect to a language definition that describes the structure of the abstract syntax tree, the transformation rules and the reduction rules. In addition a language definition may contain a collection of local value and function definitions. A language definition is written in a meta-language that is introduced by this section. The language definition is described in overview and the key features of the language are given as a series of examples that follow.

Figure 5 defines the meta-language. A language defined using **deflang** introduces the abstract syntax in the **abstract** clause, the transformation rules in the **transform** clause, and the reduction rules in the **reduce** clause. The local identifiers introduced by **locals** are scoped over the rules. An abstract-clause is a tree-structure definition, including alternatives. It is defined using mutually recursive named abstract-rules where ***** is used to express 0 or more repetitions and **or** is used to express alternatives. An editor starts with respect to a named abstract-rule causing the abstract-syntax tree to be created as an instance of the clause, up to the point where alternatives or repetition is encountered. At that point the editor must wait for user input in the form of menu selection in order to choose between the alternatives, to add another instance of a repeated clause or to delete a repeated clause. For example, given the abstract definition:

```
(abstract
  [tree (node (* (or leaf tree)))]
  [leaf (data str)])
```

and a starting clause **tree**, the editor will generate abstract syntax for **(node h)** where **h** is a *hole* requiring user input to produce another child. If the user opts to produce another child then the result is another hole that requires the user to select between a leaf or a tree. Selecting a leaf will replace the hole with a child that has the form **(data s)** for an editable string **s**. Selecting a tree, will cause the hole to be replaced with a new **(node h)** structure, causing the process to be replicated at the child level.

The abstract-clause provides guidance to the editor on the user-assisted development of the abstract syntax structure. However, it is not a type definition, since the transformation rules are free to replace any element of the abstract syntax with structures that are not defined within the clause.

Both the transformation and reduction rules are defined in terms of *rules*. A rule has a pattern and an expression. When applied to an abstract syntax tree, the rules are tried in turn: if their pattern matches the current structure then the expression is evaluated to produce a replacement structure. In both cases, the rule-sets are applied using a top-down, left-to right strategy. If the rules match then the matching sub-tree is replaced with the freshly constructed tree and the process is repeated from the root. The process terminates when there is no sub-tree that matches any rule-pattern.

If a pattern matches a syntax-tree then the result is a collection of identifier bindings that are used in the corresponding expression evaluation. If an identifier is encountered twice in the same pattern then it must be bound to the equal values, where values with the same structure are equal even if they have different identities. A pattern **(f x y z)** will match any tree with the same functor **f** and where the corresponding sub-trees match. A pattern **((f i) x y z)** will match as before, but will also match **i** against the identity of the tree. When an expression **(f x y z)** is performed, a tree with a new identity is created. The expression **((f i) x y z)** creates a tree with the designated identity **i** thereby allowing tree identities to be propagated by rules.

Syntax trees may have multiple children. The pattern **(f p ...)** matches any tree that has a root-functor **f** and for which each child matches **p**. Each variable in **p** may be matched multiple times in which case they may be used multiple times in expressions by adding **...** after an expression that refers to them. For example, the following rule: **[(node c1 ... (data "x") c2 ...) (node c1 ... c2 ...)]** may be used to remove a child from a node. Local definitions might be used to remove all children of a particular form:

```
(locals
```

```

[(remove-xs n)
 (case n
  [(node c1 ... (data "x") c2 ...) (remove-xs (node c1 ... c2 ...))]
  [(node c ...) (node (remove-xs (node c)) ...)]
  [(data s) (data s)])])
(transform
 [(node c ...) (remove-xs (node c ...))])

```

The reduction rules of a language are used to produce a normal-form that represents the concrete syntax to be displayed to the user. Figure 5 shows the definition **nf** of normal-forms. In most cases these are trees with designated functors, for example (**box** (**border** 1) (**centre** "x")). In some cases it is important that the identity of the normal-form syntax-tree is consistent for each reduction. Where this is necessary, the identity **i** can be set as above: ((**box** **i**) (**border** 1) (**centre** "x")).

Normal-forms describe how they should be displayed on the screen. In addition, boxes contain menus that appear when the box is selected. Each menu element contains a message that is sent to the current abstract syntax tree when selected. The following list provides an overview of normal-forms: **nl** produces a new-line; **seq** is used to display a sequence of elements; **tab** and **indent** are used to control text formatting; **ellipse**, **rectangle**, **image** and **thumbnail** are used to display shapes; **chars** creates a text editor. A normal-form created using **tree** contains a root normal-form and any number of children.

Boxes contain elements and are created using **box** in which case the elements are positioned relative to the top-left corner of the box, **hbox** where the elements are listed horizontally, or **vbox** where the elements are listed vertically. Boxes may have outlines and have menus as discussed above. Boxes are selectable elements and therefore it is important that they have a consistent identity for each reduction.

Graphs are created using **graph** and contain a collection of edge types, nodes and edges. The edge types determine whether edges can be drawn between designated node types when the user drags the mouse from a source node to a target node. Nodes are created as ((**node** **i**) (**t**) **j** **nf**) where **i** should be consistent over reductions and allows the editor to cache the node position, **t** is the node type, **j** is the identity of the abstract syntax tree that has been mapped to the node, and **nf** is the concrete-syntax to display on the node. Edges should also have consistent identities and include information about end decorations and labels.

The following sections provide examples of the use of the meta-language. In each case the example has been implemented in the editor, although the definitions have been reduced slightly by omitting certain details where this should not cause confusion.

5 Language Definition and Pattern-Based Reduction

Figure 6 shows a simple example of a language definition and the editor that is produced. The abstract syntax structure that is shown on the right is (**gene** (**a**) (**a**) (**c**) (**t**) (**g**) (**g**)). Since the abstract-syntax defines a gene tree to contain * **letter** the blue circle denotes a hole that can be selected using the mouse to add more letters. The red menu on the right shows the keyboard short-cuts that can be selected by pressing the key corresponding to the first letter of the option.

The concrete syntax produced by the **reduce** rules is a sequence of strings: (**seq** "a" "a" "c" "t" "g" "g" **h**) where **h** is a hole. The editor displays sequences of elements horizontally on the screen. The hole is used to capture mouse events that allow the user to create new letters.

```

1 (deflang DNA
2   (abstract
3     [DNA (gene (* letter))]
4     [letter (or (a) (c) (t) (g))])
5   (reduce
6     [(gene letter ...) (seq letter ...)]
7     [(a) "A"]
8     [(c) "C"]
9     [(t) "T"]
10    [(g) "G"]]))

```

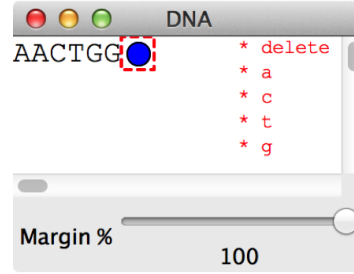


Fig. 6: Sequences of DNA

```

1 (deflang boxes
2   (abstract [root (root (boxes) tree)] [tree (or (tree str (* tree)) (leaf str))])
3   (transform
4     [(send (root _ t) (key-pressed _ #\t)) (root (tree) t)]
5     [(send (root _ t) (key-pressed _ #\b)) (root (boxes) t)])
6   (reduce
7     [(root (boxes) t) (tree->boxes t)]
8     [(root (tree) t) (tree->tree t)]
9     [(tree->boxes (tree data child ...))
10      (hbox (outline 1) (centre data) (align (vbox (outline 1) (align (tree->boxes child) ...)))])
11     [(tree->boxes (leaf d)) d]
12     [(tree->tree (tree data child ...)) (tree data (tree->tree child) ...)]
13     [(tree->tree (leaf d)) d]))

```

Fig. 7: Trees Displayed in Two Modes

6 Boxes and Trees

Concrete syntax needs to structure information so that it can be presented to users in a suitable way. The previous section showed how `seq` is used to linearise information. Boxes can be used to define groups of concrete syntax elements that to organise them horizontally and vertically. Boxes conveniently nest so that large scale structures can be organised. Another way to organise structured information is to use trees. A tree has a data element at the root and contains a sequence of elements as children. Trees can be nested to any depth.

Decision trees can be organised using nested boxes or trees. Consider the following abstract syntax tree:

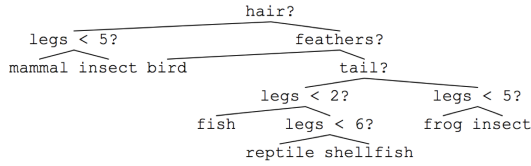
```

(root (tree)
  (tree "hair?"
    (tree "legs < 5?" (leaf "mammal") (leaf "insect"))
    (tree "feathers?"
      (leaf "bird")
      (tree "tail?"
        (tree "legs < 2?"
          (leaf "fish")
          (tree "legs < 6?" (leaf "reptile") (leaf "shellfish")))
        (tree "legs < 5?" (leaf "frog") (leaf "insect"))))))))

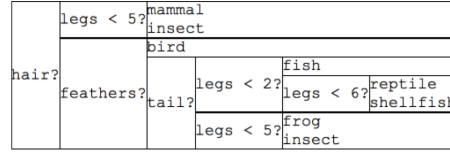
```

The tree is used to determine the type of an animal. The idea is that the decision making process starts at the root with the question *does the animal have hair?*. If the answer is yes, then move on to the first sub-tree, otherwise move to the second sub-tree. If the process ever reaches a leaf, the animal has been categorised.

The tree can be represented using nested boxes by representing a tree-node as a horizontal box with the question on the left and the sub-trees on the right. Each sub-tree is listed one above the other so that if the answer to the question is *yes* then the first sub-tree is selected.



(a) Displaying Decisions as a Tree



(b) Displaying Decisions as Nested Boxes

Fig. 8: Displaying a Decision tree

Figure 7 shows the definition of the decision tree language. The root node of the abstract syntax contains an element (**boxes**) or (**tree**) that is used to mode the reduction rules. The reduction rule **tree->boxes** is used to map the abstract syntax to nested boxes whereas the rule **tree->tree** is used to produce nested trees.

Boxes may specify an outline using the optional (**outline n**) element where **n** is the width of the line. Each element of a box must be supplied with a position. In the example the question is centred the **hbox** whereas the sub-trees are aligned.

The definition of the reduction rules in figure 7 provides an example of how rules process recursively defined structures. Notice that the definition of rules **tree->boxes** and **tree->tree** uses both of these functors in the expression part of the rule. This should not be interpreted as a recursive *call* to the rule: the expression is creating a new abstract syntax structure with the supplied functor. The operational semantics of rule matching means that if a rule is applied to a tree then all rules are re-applied to the root of the tree. Therefore the effect is a recursive call to the rule.

Figure 7 shows how transformation rules are used to process events. When the user presses a key —tt k, the message (**send s (key-pressed i k)**) is sent to the abstract syntax tree. In effect this is done by updating the tree to become a new tree and then running the mappings. Transformation rules can make changes to the abstract syntax tree managed by the editor and therefore is used to consume messages sent to the tree. The transformation rules in the example, show that the keys are consumed and cause the mode on the root of the abstract syntax tree to be changed.

Figure 8 shows two screenshots of the editor. The first shows the example tree where the mode is (**tree**) and the second where the mode is (**boxes**). The two modes are toggled using the **t** and **b** keys.

7 Transformations and Evaluation

Transformation rules make changes to the abstract syntax tree before it is reduced. This feature of the approach can be used to implement an operational semantics for a language. Consider a simple λ -calculus and a step-wise reduction strategy. A convenient concrete representation for such a calculus is a tree that is rooted either at an application node, at a λ , or at an atom (identifier or constant). In order to make the language more useful we also add pairs.

A λ -expression is in a *normal-form* if it contains no applications. An evaluation step performs a single application producing a new expression. The calculus is defined as a language in figure 9. A λ -calculus provides support for recursive definitions using the **Y**-combinator. This is a function that can be applied to a function and return the fixed-point of the function, *i.e.*, $f(Y(f))=Y(f)$. Suppose that we define a function $f = \lambda L. (1,L)$, then $Y(f)$ is the infinite

```

1 (deflang lambda-calculus
2   (abstract
3     [exp (or
4       (const str)
5       (pair exp exp)
6       (ident str)
7       (apply exp exp)
8       (lambda str exp))])
9
10  (transform
11    [(send t (key-pressed _ #\e))      (eval-step t)]
12
13    [(eval-step (lambda arg exp))      (lambda arg (eval-step exp))]
14    [(eval-step (pair exp1 exp2))      (pair (eval-step exp1) (eval-step exp2))]
15    [(eval-step exp)                   (eval exp)]
16
17    [(eval (apply (lambda arg body) exp)) (subst exp arg body)]
18    [(eval (apply exp1 exp2))             (apply (eval exp1) exp2)]
19    [(eval exp)                          exp]
20
21    [(subst new old (const k))           (const k)]
22    [(subst new old (pair first second)) (pair (subst new old first) (subst new old second))]
23    [(subst new old (ident old))         new]
24    [(subst _ old (ident name))          (ident name)]
25    [(subst new old (apply exp1 exp2))   (apply (subst new old exp1) (subst new old exp2))]
26    [(subst new old (lambda old exp))    (lambda old exp)]
27    [(subst new old (lambda arg exp))    (lambda arg (subst new old exp))])
28
29  (reduce
30    [(hole _) h] h]
31    [(ident s) s]
32    [(const k) k]
33    [(pair e1 e2) (tree "." e1 e2)]
34    [(apply e1 e2) (tree "@ " e1 e2)]
35    [(lambda i e) (tree (hbox (fixed) (align "λ") (align i)) e))])

```

Fig. 9: λ -calculus Evaluation

sequence of 1s: $(1, (1, (1, \dots)))$. Figure 10 shows the definition of Y as abstract syntax on the left and editor screen-shots starting with `(apply Y (lambda "L" (pair (const "1") (ident "1"))))`. The starting state is shown in the centre of the spiral and each subsequent state is an evaluation-step. The execution shows how the Y-combinator gradually emits 1s on the left-hand side of the tree.

8 Graphs

Many different languages lend themselves to concrete representations in the form of graphs. Class diagrams place classes on nodes and associations on edges. State machines place states on nodes and transitions on edges. Maps show locations on nodes and routes between locations on edges.

The meta-language provides support for graphs by translating abstract syntax to a normal-form: `(graph (edge-types edge-type ...) node ... edge ...)` Edge-types are defined to control what happens when a user drags an edge between nodes. Graph nodes are created with a designated type and not all types of node can be connected with edges; furthermore, the same two node types may be connected with different types of edge. For example, on a use-case diagram the edge-type *uses* holds between a node of type *actor* and a node of type

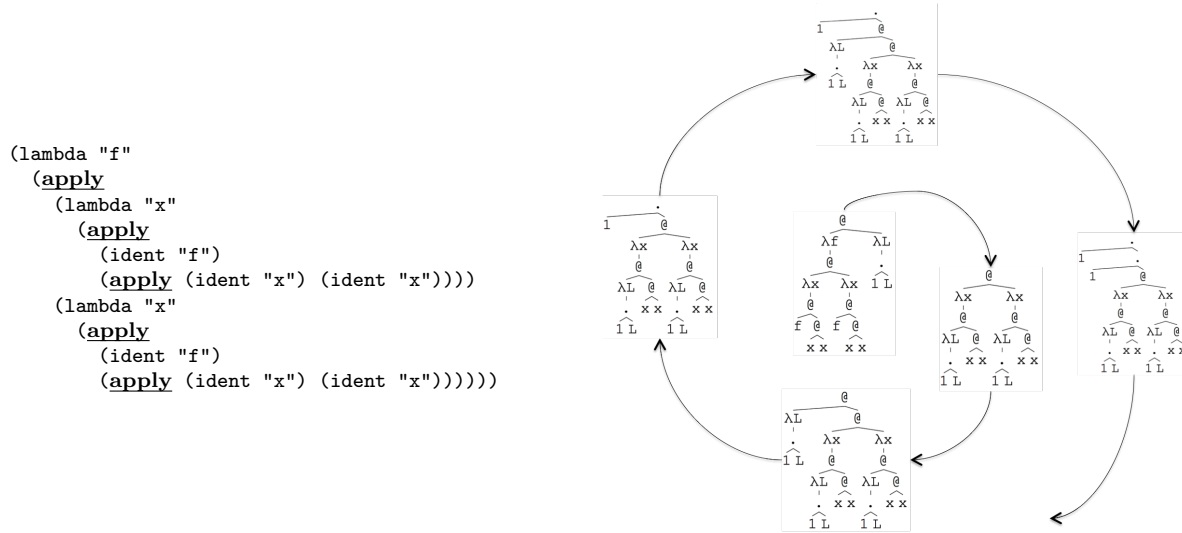


Fig. 10: Definition of Y and evaluation of $Y(\lambda L. .(1,L))$

use-case but not vice versa. In addition, edges of types *extends* and *includes* may both hold between the same nodes of type *use-case*.

Both edge-types and node-types can be any value, but are conveniently expressed as 0-ary trees whose functor designates the type. Suppose that we want to create a use-case graph with a single actor and two use-cases. The actor uses both use-cases and one use-case includes and extends the other (for the sake of the example). The graph can be expressed as follows (assuming appropriate identities and displays):

```
(graph
  (edge-types
    (uses (actor) (use-case))
    (includes (use-case) (use-case))
    (extends (use-case) (use-case)))
  (node (actor) id1 actor-display)
  (node (use-case) id2 use-case-display)
  (node (use-case) id3 use-case-display)
  (edge (uses) id1 (none) id2 (arrow))
  (edge (uses) id1 (none) id3 (arrow))
  (edge (includes) id2 (none) id3 (arrow) (label (target) "<<includes>>"))
  (edge (includes) id2 (none) id3 (arrow) (label (target) "<<extends>>")))
```

Graphs can contain arbitrarily complex displays on the nodes and the edge-labels. As an example of this consider the case where an abstract syntax structure is defined to be recursive in terms of the data on the node, *i.e.*, graphs are nested on nodes. This type of structure occurs in nested state-charts, for example. We would like to construct an editor that allows a graph to be constructed in terms of nodes and edges and then to allow a particular node to be selected by double-click so that the editor switches focus and displays the graph that is *on the node*. Of course this process can continue to any level of nesting. If the user wishes to move from a child graph to a parent graph then the up-arrow key is used to return from the most recent double-click.

The requirement that the editor can move from parent to a descendent at any level and then return incrementally back through the levels, implies that the editor must have a state that is used to record the current level of nesting. This can easily be achieved with a projectional editor because the abstract syntax structure can contain any amount of information that is ignored by the reduction mapping. The state of the example editor is to be represented as

```

1 (deflang nested-graph
2   (abstract
3     [machine (machine G (empty))]
4     [G (graph (entities (* entity)) (relationships))]
5     [entity (entity G)])
6   (transform
7     [(send (machine (graph (entities e1 ... ((entity i) g) e2 ...) r) dump) (double-click (list _ i)))
8      (machine g (dump id (graph (entities e1 ... e2 ...) r) dump)) ]
9     [(send (machine g (dump i (graph (entities e ...) r) d)) (key-pressed _ "up"))
10      (machine (graph (entities ((entity i) g) e ...) r) d)]
11     [(send (machine (graph n (relationships r ...)) d) (new-edge type source target))
12      (machine (graph n (relationships (relationship source target) r ...) d)]]
13   (reduce
14     [(machine g _) (->graph g)]
15     [(->thumbnail (graph (entities e ... _) (relationships r ...)))
16      (thumbnail 0 0 40 40 (graph (edge-types (edge (entity) (entity))) (->node e) ... (->edge r) ...))]
17     [(->graph (graph (entities e ...) (relationships r ...)))
18      (graph (edge-types (edge (entity) (entity))) (->node e) ... (->edge r) ...)]
19     [(->node ((hole i) h)) ((node "n" i) (entity) i h)]
20     [(->node ((entity i) g))
21      ((node "n" i) (entity) i ((box "b" i) (align (ellipse 0 0 50 50 #f #f)) (align (->thumbnail g)))]
22     [(->edge ((relationship r) source target)) ((edge r) source (none) target (arrow))]]])

```

Fig. 12: Nested Graphs

(machine graph dump) where **graph** is the current graph to be displayed in concrete syntax and **dump** is a stack of machine states that is used when descending and ascending the nested graph structure.

A typical abstract-syntax machine-state is:

```
(machine (graph ((entity i) g1) ((entity j) g2) (relationship g1 g2)) d)
```

for some nested graphs **g1** and **g2**, and some dump **d**. The abstract-syntax graph is reduced to a concrete syntax-graph by mapping entities to nodes and relationships to edges. If the user double-clicks on **g1** then the machine becomes:

```
(machine g1 (dump i (graph ((entity j) g2) (relationship g1 g2)) d))
```

Note that the selected node is saved on the dump in terms of its identity and its graph. Note also, however, that the saved dumped graph omits the selected node since this must be restored later. For example, if after several edits the abstract-syntax tree has become:

```
(machine g1' (dump i (graph ((entity j) g2) (relationship g1 g2)) d))
```

then pressing up-arrow results in a new machine state:

```
(machine (graph ((entity i) g1') ((entity j) g2) (relationship g1 g2)) d)
```

In addition to the behaviour described above, we might want to display a graph in the editor so that thumbnails of nested graphs are shown inside the appropriate nodes. This might be used to help the user navigate through the structure. Figure 11 is a screen-shot of the editor showing several nodes drawn as ellipses containing a thumbnail of the graph nested on the node.

The language definition for nested graphs is shown in figure 11. The **transform** clause contains rules that process the events that can occur on the editor:

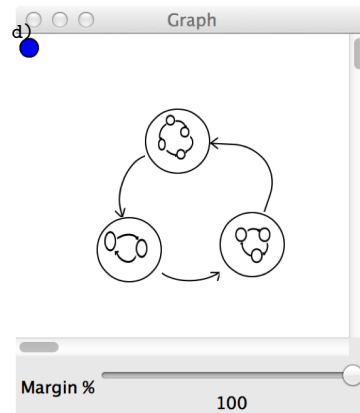


Fig. 11: Nested Graphs

- (**double-click i**) This event occurs when the user selected an element. The supplied value *i* is the identity of the selected element. In order to select a graph node, the node must contain a selectable element as part of its display. Boxes are selectable elements and are designated an identity (**list "b" i**) where *i* is the identity of the abstract-syntax entity. Therefore, the selected entity can be picked out using pattern matching as shown in the first transformation rule.
- (**key-pressed i k**) This event is supplied when the user presses a key and the element with identity *i* is selected (*-1* is supplied when nothing is selected). This is used in the second rule to pop the dump.
- (**new-edge type source target**) This event occurs when the user drags an edge from one node to another. If the nodes cannot be associated because there is no appropriate edge type defined then the event is not generated. Otherwise, if there are multiple edge types between the selected nodes then the user is required to choose via a menu. Once the event is generated, the **source** and **target** arguments are the identities of the abstract-syntax elements corresponding to the concrete-syntax nodes. The example shows a new relationship being created as a result of this event that is subsequently transformed into a concrete-syntax edge.

The graph example above does not contain any data on the nodes. A typical use of graphs for modelling languages is to place data on the nodes and to represent relationships between the data using edges and labels. Suppose that we want to define a class-based modelling language. A typical concrete-syntax for such a language is to show classes as boxes containing names and field information. Associations between classes are shown as edges between class-nodes.

Now suppose that we want to *mode* the information on the nodes by allowing the user to select the appropriate way in which the information should be displayed on a node-by-node basis. A simple example of this might be to show classes using a textual syntax. The resulting requirement is for a modelling language that can mix graphical with textual representations as shown in figure 13. The screen-shot on the left shows a simple data model displayed using a conventional concrete-syntax. The screen-shot on the right shows the same model where some of the classes have been switched to a text-based representation. Although the nodes contain text, the model is still a graph that allows the nodes to be freely moved around.

The language definition for mixed-mode class diagrams is shown in figure 14. The language is explained as follows. The abstract-syntax definition represents a model as a collection of classes and associations. The classes can be created via a hole that will be displayed as a blue-ellipse in the editor. The associations are created via **new-edge** events. Classes contain a

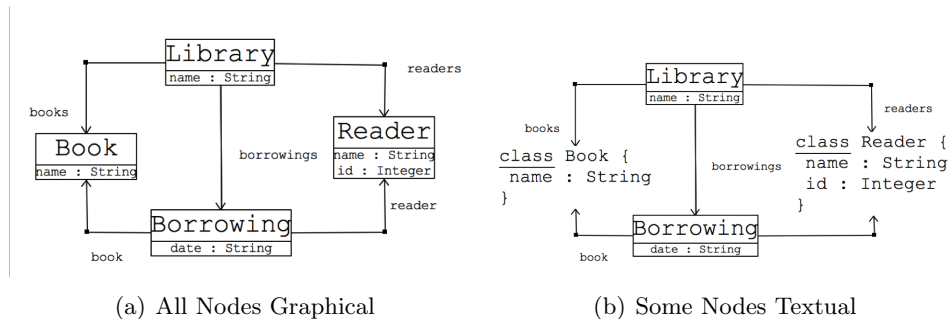


Fig. 13: Mixed Mode Class Models

```

1 (deflang class-models
2   (abstract
3     [model (model (classes (* class)) (assocs))]
4     [class (class (graphical) str (* field))]
5     [field (field str type)]
6     [type (or (string) (integer) (boolean))])
7   (transform
8     [(send (model (classes c1 ... ((class i) m n f ...) c2 ...) assocs) (double-click (list _ i)))
9      (change-mode ((class i) m n f ...) (classes c1 ... c2 ...) assocs)]
10    [(send (model cs (assocs a ...)) (new-edge (assoc) s t))
11      (model cs (assocs (assoc str s t) a ...))])
12   (reduce
13     [(model (classes c ...) (assocs a ...))
14      (graph (edge-types (assoc (class) (class))) (class->node c) ... (assoc->edge a) ...)]
15     [(class->node ((hole i) h)) ((node "create-class" i) (new-class) i h)]
16     [(class->node ((class i) (graphical) name field ...)) (class-g-node i name field ...)]
17     [(class->node ((class i) (textual) name field ...)) (class-t-node i name field ...)]
18     [(assoc->edge ((assoc i) n sid tid))
19      ((edge i) sid (none) tid (arrow) ((label "l" i) (target) (down-font 3 n))))])

```

Fig. 14: A Language for Mixed Mode Class Diagrams

name and a sequence of fields, and each field has a name and a type which is selected, again via a hole, from string, integer and boolean.

When a class is selected via mouse double-click, a mode change is performed using the following local:

```

[(change-mode ((class i) mode name field ...) (classes c ...) assocs)
 (case mode
  [(textual) (model (classes ((class i) (graphical) name field ...) c ...) assocs)]
  [(graphical) (model (classes ((class i) (textual) name field ...) c ...) assocs))])

```

When a **new-edge** event occurs, a new association is created. note that the name of the association is freshly created.

Reduction of a model creates a new graph whose nodes are mapped from classes and whose edges are mapped from associations. Note that a class node might be the hole that is used to create new classes and therefore this is caught using pattern matching and translated to a node of type **new-class** whose display is the hole (with the creation menu). A class may exist in one of two modes that affect how the nodes are created using following locals:

```

[(down-font n d) (case n [0 d] [_ (font (-) (down-font (- n 1) d))])]

[(class-g-node i name field ...)
 ((node "n" i) (class) i
  ((vbox "b" i) (fixed) (outline 1)
   (centre (up-font 3 name))
   (align ((vbox "v" i) (outline 1) (centre (down-font 3 (field-box field))) ...))))])

[(type-name t)
 (case t [(string) "String"] [(integer) "Integer"] [(boolean) "Boolean"] [(hole _) h] h))]

[(field-box f)
 (case f
  [(field i) name type) ((hbox i) (fixed) (align name) (align " : ") (align (type-name type)))]
  [(hole i) h] h))]

[(class-t-node i name field ...)
 ((node "n" i) (class) i
  ((box "b" i) (fixed)
   (align (tab (indent 10)
    (seq (underline "class") (space) name " {" (nl)
      (vbox (fixed) (align (field-box field)) ...)) (nl) "}")
    ))))])

```

9 Implementation and Examples

Section 2 introduced a requirement for a language that could be used to both construct (model) and play (simulate) a game. The previous sections have used a variety of example languages to introduce the features of a projectional editor that are necessary to support the game. Size limitations prevent the full implementation of the game to be included in this paper, however it is included the implementation pack that accompanies the paper¹. The pack includes a Mac disk image `stand-alone-editor.dmg` of the editor implementation, several saved languages (*.xml) and the source code of the language definitions `language-definitions.rkt`. Once you have installed the editor, navigate to the `bin` directory and start the tool before dragging any of the xml files onto the editor pane to load up the language definition. The pack includes the game language definition, an example adventure, a use-cases implementation of hotel booking and a library class diagram. See the language definitions for more details.

10 Analysis, Deficiencies and Future Directions

This paper has introduced an architecture for projectional editors that is defined in terms of a loop processes abstract and concrete-syntax with respect to an editor that has a display and event interface. In addition the paper proposes a meta-language that can be used to initialise and drive the editor for a range of different concrete languages. The approach and meta-language has been implemented in Racket and the paper contains a number of examples that whose language definitions and screen-shots have been taken directly from the implementation.

A projectional approach allows the use of a language to be *modal*. This is quite easy to achieve because the mode can be a function of the abstract-syntax that can be projected onto an appropriate concrete-syntax. Examples in this paper include the class models and the game. A declarative meta-language for abstract-syntax definition allows an editor to provide automatic support for the creation of elements and the choice between alternative elements. In the case of the game, a menu that allows new rooms to be created is auto-generated. Furthermore, by reifying the choice-point, the menus can be offered within the concrete-syntax, allowing the interactions to be context-specific. A suitably equipped projectional editor can easily provide mixed textual and graphical interaction modes. In the case of the technology described in this paper, graphs, trees and text are all specialisations of a more general type of concrete-syntax element and can therefore be arbitrarily interleaved.

Pattern-directed mapping rules are a very convenient way of processing syntax, which is, after all, just a data structure. It is possible to identify different categories of mapping rule that are useful in this regard. Firstly, there are transformations that are used to change the state of the abstract-syntax. These are important in order to allow user-events to be handled in a consistent way. In addition, such rules can be used to add an operational semantics to a language without introducing any new features to the meta-language. Secondly, there are mappings that take the current abstract-syntax structure and project it onto a concrete-syntax structure that corresponds to the display interface of an editor. as has been shown, these can be achieved using pattern-directed rules and therefore require no new meta-language machinery. The range of concrete-syntax elements will depend upon the application and the sophistication of the editor, however the selection shown in this paper seems sufficient for a wide range of language-applications and could easily be extended.

¹ http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/projectional_editor_demo.zip

Identity-management is an important feature of projectional-editing; it is required to manage the association between abstract and concrete-syntax so that events occurring in the context of a concrete-element can be associated with a corresponding abstract-element. Secondly, there is information relating to the concrete-syntax that has no counterpart in the abstract-syntax, for example the position of graph-nodes or the number of way-points on an edge. In principle, the concrete-syntax is re-generated from the abstract-syntax each time the user performs an edit and therefore, if the identities are not managed appropriately the layout information would be lost.

Although the approach and corresponding implementation have been shown to support a range of interesting and useful languages, a number of deficiencies remain to be addressed. A criticism of projectional technology is that it tends to be syntax-directed which can reduce usability by forcing the user to enter language structures through a pre-defined interface. One possibility is to take a light-touch approach whereby the user is only aware of the structure behind the concrete-syntax when they either ask to be told or when it becomes helpful to do so. To the user it appears as though they are working with a text-based editor with a knowledgeable expert looking over their shoulder occasionally prompting them or to whom they can refer when things get tricky. This approach is fine when dealing with certain types of language, for example programming languages. However, projectional technology is arguably not the best support for programmers whose mental models of their systems are (or ought to be) far more sophisticated, dynamic and responsive than any projectional technology could ever hope to be. Perhaps projectional technologies might best be applied in cases where there is obviously a need for structural support, for example when zooming in and out of structures that need to be related or when there is a repeated pattern of structure to be completed. In addition, it is possible that different styles of editor-interaction are required within the same system development: projectional-style for the big-picture stuff and more traditional-style (albeit with knowledgeable support) for the detail. In any case, it is not clear which approach is more appropriate and the case has not been made that any one approach is universal, therefore this paper is claimed to be a valid contribution to the debate.

One possible future area of investigation is to use grammar-ware at the leaves of the syntax-tree and to integrate both free-text editing and syntax-directed editing at that level. A fruitful strategy might be to allow text-boxes to be associated with grammars that synthesize abstract-syntax. If the grammar can be successfully used to parse the text then the result is subject to projectional mapping. Otherwise the content remains free text. This strategy would support mode-shifts that address the oft cited problem of modifying the internal leaves of a binary expression tree by inserting new operators without resorting to syntax-directed menus.

The meta-language described in this paper provides no support for error handling and will simply go wrong if the rules fail to produce a normal-form or if a local rule definition produces a tree of an unexpected type. One way to address this is to have a separate category of rules that are used for checking and error reporting. If the editor provides a separate error interface that can be used for bringing error messages to the attention of the user and for linking the errors to the concrete syntax then the same pattern-based rule technology can be used to perform error handling. Related to this, the language does not support static checking. For example, it should be possible to detect the use of unbound identifiers and undefined functors. Some aspects of static checking should be easy to achieve, however it would also be desirable to define a type system so that the use of syntax structures can be checked before use. It is

not clear whether it is possible to determine if the reduction rules can be shown to terminate with a valid normal-form, or under what conditions this is possible.

There is interest in the modularity and composition of languages and DSLs in particular [27, 31, 4, 5, 21, 22]. A key challenge to achieving engineered integration is posed by concrete-syntax. By inverting the focus of attention to abstract-syntax, a projectional editor does not suffer from such problems. However, there are still significant issues to be addressed and this could be a fruitful area for future work.

The current implementation has an XML-based save and load mechanism that is based on an obvious encoding of the syntax trees defined in figure 4. Given the universality of the data representation it should be possible to import data in any well-defined format, such as XMI or EMF, in order to create tool-chains.

References

1. Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In *Applications of Graph Transformations with Industrial Relevance*, pages 82–97. Springer, 2008.
2. Colin Atkinson and Ralph Gerbig. Harmonizing textual and graphical visualizations of domain specific models. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, pages 32–41. ACM, 2013.
3. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008.
4. Walter Cazzola and Davide Poletti. Dsl evolution through composition. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, page 6. ACM, 2010.
5. Walter Cazzola and Ivan Speziale. Sectional domain specific languages. In *Proceedings of the 4th workshop on Domain-specific aspect languages*. ACM, 2009.
6. James R Cordy, Thomas R Dean, Andrew J Malton, and Kevin A Schneider. Source transformation in software engineering using the txl transformation system. *Information and Software Technology*, 44(13):827–837, 2002.
7. Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, 253(7):105–120, 2010.
8. Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.
9. Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
10. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2009.
11. Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language evolution in practice: The history of gmf. In *Software Language Engineering*, pages 3–22. Springer, 2010.
12. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.
13. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254. ACM, 2006.
14. Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.
15. Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.
16. Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In *Model Driven Engineering Languages and Systems*. Springer, 2010.

17. Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Fiona AC Polack. Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 13–19. IEEE Computer Society, 2009.
18. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.
19. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.
20. Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
21. Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language boxes. In *Software Language Engineering*, pages 274–293. Springer, 2010.
22. Bernhard Rumpe. Towards model and language composition. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. ACM, 2013.
23. Markus Scheidgen. Textual modelling embedded into graphical modelling. In *Model Driven Architecture—Foundations and Applications*, pages 153–168. Springer, 2008.
24. Christian Schneider. On integrating graphical and textual modeling. *Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel*, 2011.
25. Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *ACM SIGPLAN Notices*, volume 41, pages 451–464. ACM, 2006.
26. Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.
27. Federico Tomassetti, Antonio Vetró, Marco Torchiano, Markus Voelter, and Bernd Kolb. A model-based approach to language integration. In *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*. IEEE, 2013.
28. Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.
29. Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1449–1450. IEEE, 2012.
30. Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 2010.
31. Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. Improving domain-specific language reuse with software product line techniques. *Software, IEEE*, 26(4):47–53, 2009.